Nagging: A scalable fault-tolerant paradigm for distributed search

Alberto Maria Segre [a,*], Sean Forman [b], Giovanni Resta [c],
Andrew Wildenberg [d]

[a] *Department of Management Sciences, The University of Iowa, Iowa City, IA 52242, USA*
[b] *Mathematics and Computer Science Department, Saint Joseph's University, Philadelphia, PA 19131, USA*
[c] *Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, I-56124 Pisa, Italy*
[d] *Computer Science Department, SUNY Stony Brook, Stony Brook, NY 11794, USA*

Abstract

This paper describes *nagging*, a technique for parallelizing search in a heterogeneous distributed computing environment. Nagging exploits the speedup anomaly often observed when parallelizing problems by playing multiple reformulations of the problem or portions of the problem against each other. Nagging is both fault tolerant and robust to long message latencies. In this paper, we show how nagging can be used to parallelize several different algorithms drawn from the artificial intelligence literature, and describe how nagging can be combined with partitioning, the more traditional search parallelization strategy. We present a theoretical analysis of the advantage of nagging with respect to partitioning, and give empirical results obtained on a cluster of 64 processors that demonstrate nagging's effectiveness and scalability as applied to A^* search, $\alpha\beta$ minimax game tree search, and the Davis–Putnam algorithm.
© 2002 Published by Elsevier Science B.V.

1. Introduction

Many artificial intelligence problems of practical interest can be posed in terms of search. Not surprisingly, the development of a robust network infrastructure coupled

---

* Corresponding author.
 E-mail addresses: segre@cs.uiowa.edu (A.M. Segre), sforman@sju.edu (S. Forman), resta@iit.cnr.it (G. Resta), awilden@cs.sunysb.edu (A. Wildenberg).

with the advent of reasonably-priced computing equipment has helped focus attention on techniques that use multiple processors operating in parallel to improve search performance. Some of this work has centered on developing application-level toolkits to access a distributed computing environment as well as resource-management tools that enable an application to exploit computing resources that span organizational boundaries (e.g., "grid computing") [2,11,15,22].

Aside from differences in enabling technology, most attempts to parallelize search are quite similar, involving some sort of *partitioning* [13,14]. The general idea is that each processing element takes responsibility for a portion of the search space, and that the individual solutions to these subproblems can then be compared or composed to obtain a solution to the original problem. Different partitioning schemes usually differ in their target computer architecture (e.g., SIMD vs. MIMD, shared memory multiprocessors vs. networks of workstations, etc.) and how they address a number of technical problems, such as *load balancing* (how best to assign work in order to exploit all available processors all of the time) and *fault tolerance* (how to notice and recover from the momentary inaccessibility or outright loss of one or more processing elements). Nevertheless, partitioning is the basis of a diverse set of projects, including SETI@Home, the search for radio signal evidence of extraterrestrial life, GIMPS, the search for Mersenne prime numbers, and various code breaking efforts from distributed.net.

Unfortunately, many problems do not partition well. One reason is that information acquired early in a serial search process can often be used to reduce the amount of search performed overall: for example, consider how the α and β values are used to prune the game tree in $\alpha\beta$ minimax search. If the search is partitioned, the information acquired while searching one subspace may come too late to help reduce the search in another subspace explored simultaneously, resulting in a search that may actually be slower with multiple processors than it is with a single processor.[1] This problem is an instance of the more general *speedup anomaly* problem, first studied for branch-and-bound style algorithms on two well-known NP-hard problems in [19]. A speedup anomaly occurs when a solution is obtained more slowly with more processors than it is with fewer processors, or, alternatively, when a solution is obtained superlinearly faster with multiple processors. Later, such superlinear speedups were routinely studied, particularly within the parallel logic programming and theorem proving communities (see, e.g., [7]).

This paper describes a distributed search paradigm called *nagging* that exploits the speedup anomaly often observed when parallelizing problems by playing multiple reformulations of the problem or portions of the problem against each other. Nagging has several advantages over partitioning techniques; it is intrinsically fault tolerant, naturally

---

[1] Even if the information were made available in a timely fashion, sharing the information among multiple processors would entail some amount of communication overhead. In general, the cost of communication tends to increase as the number of processors increases. Depending on the underlying architecture, sharing information may involve interprocessor communication or the use of shared memory. For shared-memory multiprocessors, there are practical design limits on the number of processors one can incorporate in a single machine. For more loosely-coupled processors, interprocessor communication (either in directed or broadcast form) requires overhead for generating and servicing messages; furthermore, as the number of processors increases, higher message latencies associated with larger networks generally entail larger communication overheads.

load-balancing, requires relatively brief and infrequent interprocessor communication, and is robust in the presence of reasonably large message latencies. These properties help make nagging suitable for use on geographically-distributed networks of processing elements. Originally developed in the course of our work on distributed automated deduction [35,39, 40], we show here how nagging can be generalized and applied to a broad range of search algorithms from the artificial intelligence literature. We develop an analytical performance model comparing nagging and partitioning, and use this model to make some predictions about their respective performance. Finally, our performance claims are justified via an empirical evaluation of nagging and partitioning on three well-known yet significantly different search algorithms; A^* search [20], $\alpha\beta$ minimax game tree search [27], and the Davis–Putnam search algorithm [6].

2. Nagging and search

Nagging is an asynchronous parallel search pruning technique where a single *master processor* (or *master*), performing some standard search procedure, is advised by one or more *nagging processors* (or *naggers*), each performing identical search procedures, about portions of its master's search space that need not be explored.

Let us consider a search procedure that is designed to find a globally-optimal solution in a finite, implicitly-defined, search space; this is the case, for example, when trying to determine the best next move to make in game tree search with fixed horizon (similar arguments hold for situations where any legal solution suffices, such as for theorem proving or satisfiability problems). At initialization, each nagger obtains the problem specification from its master processor. It then engages in a series of *nagging episodes*, each initiated by the nagger when it becomes idle, until the master has completed its search.

A nagging episode begins when the nagger requests a snapshot of its master's current state, which is concisely described by communicating the sequence of choices made through the predefined search space (see Fig. 1). The master selects a *nagpoint* (a node along its own current path from which the nagger will begin its own exploration of the space) according to some predefined nagpoint selection criteria; it then communicates this nagpoint and the value describing its own current best solution to the nagger.

The master and the nagger now race to exhaust their respective search spaces; while the two search spaces are semantically equivalent, they may well be searched differently, leading to different expected solution times. If we are searching for an optimal solution, there are only three possible outcomes to consider:

(1) *Abort*: If the master backtracks over the nagpoint before the nagger completes its own search, the master signals the nagger to abort the nagging episode, which causes the nagger to once again become idle and initiate a new nagging episode.
(2) *Prune*: If the nagger completes its search and finds there are no better solutions than that already known by the master, then the nagger interrupts the master and forces it to backtrack to the nagpoint, resulting in a reduction in the master's search space.
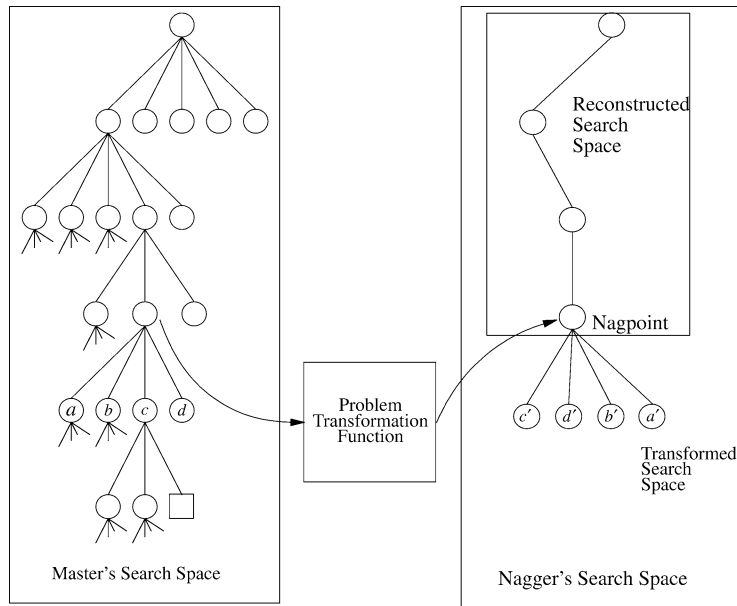
Fig. 1. Nagging episode. The square node indicates the current position of the master process, which is executing a depth-first search. A nagpoint is selected along the master's search path, and is described to the nagging process by communicating the series of choices from the root of the search tree to the nagpoint. The nagger reconstructs the master's search space up to the nagpoint and then commences exploring its own, transformed, version of the space rooted at the nagpoint.

(3) *Solve*: Finally, should the nagger find a better solution than that communicated by the master, it can abort its own current search and report the new value to the master so that the master might use this new value to reduce its own search space.

Of course, for nagging to have the greatest possible positive effect on the master's search efficiency, we would prefer the second and third cases occur with high probability, and we hope the first case occurs only rarely. Should the second and third cases never occur, we can expect no improvement in the master's expected time to solution: indeed, the master's search will surely be less efficient due to the small—but measurable—additional overhead of servicing its naggers.

Two techniques are applied to improve the odds. First, a nagger can be nagged recursively by yet another processor in order to help it exhaust its own search space more quickly. Second, and more importantly, each nagger may apply a *problem transformation function* to reduce, in practice, the size of its search space while retaining at least some of the information content implicit therein (note that effective problem transformation functions are typically dependent on both the search algorithm in use and the problem domain itself). We will discuss both of these techniques later in this paper.

The main characteristics of nagging are clear even within this simplified context. First, nagging does not require explicit load balancing, since idle machines always initiate nagging episodes to keep themselves busy. As long as the master processor is

still searching, new nagpoints can be provided at will. Second, nagging is intrinsically fault tolerant: since the master's search is unaffected should a nagger fail or become inaccessible, losing a nagger will never compromise the correctness or quality of the solution produced by the master, nor will the master ever stop and wait for the missing nagger. Finally, communication is brief, since even deep search states can be described concisely as a sequence of choices, while the results reported by a nagger often reduce to a single bit (e.g., prune/don't prune) or just a few bits (e.g., a new value for the best solution to date).

3. NICE: A Network Infrastructure for Combinatorial Exploration

To support our work on nagging, we have developed *NICE*, the *Network Infrastructure for Combinatorial Exploration*. NICE is specifically designed to support both nagging and partitioning for search algorithms; it allows application programmers to parallelize their search procedures by making appropriate function calls within their code. The NICE distribution includes *niced*, a Unix resource management daemon, *niceq*, a daemon status query program, and *niceapi*, the applications programmer's interface library. The code is written in ANSI C with BSD sockets over TCP/IP, and is known to run on multiple variants of the Unix operating system, including Linux, Sun OS, and HPUX.

The NICE daemon, or *niced*, must be running on every participating machine, whether master or nagger. Typically, it is started automatically as part of the system boot process, and runs as long as its host CPU is running (the daemon itself is single threaded and extremely lightweight, having no noticeable impact on system performance). NICE daemons are arranged hierarchically, with each daemon reporting to a single parent daemon while answering to zero or more child daemons. The NICE daemon fulfills three primary functions:

(1) The daemon maintains contact with the NICE hierarchy, occasionally exchanging host load and availability information with its parent and child daemons, and managing failure recovery should its parent and/or child daemons become unreachable or unresponsive. Each daemon may also initiate local reorganization of the NICE hierarchy in a greedy attempt to enhance overall search performance according to each host's current actual load.
(2) The daemon provides the interface through which a qualifying application may request additional processors. It is also responsible for security, certifying which applications on which hosts are allowed to request support from other processors, which executables can be run on the local host, as well as which files, if any, can be accessed locally.
(3) The daemon manages the local host's resources according to prespecified host-specific constraints: for example, some hosts may be available only during specified times, such as during nighttime hours. Thus the daemon must decide when a processor can respond to requests for new processes, and must also manage previously spawned but still running processes, putting them to sleep and later waking them when the host becomes available once more.

A NICE-enabled application communicates with the NICE infrastructure via a set of callable functions contained in the NICE applications programmer's interface. This linkable library contains functions that, when invoked, request new copies of the application be spawned on other machines. It also contains functions to support communication between applications and between the application and the NICE daemon. Note that the library does not actually contain code for the search algorithms, but rather only the handful of functions that are needed to parallelize—via either nagging or partitioning—appropriately designed serial search algorithms.

Both the NICE daemon and NICE API represent fairly mature software efforts: versions of the NICE daemon have been running continuously on our systems for several years, with no noticeable impact on performance. The code is very robust, with NICE daemons running reliably and unobtrusively for periods of many months between system restarts. But more to the point, while the NICE infrastructure represents a necessary enabling technology that directly supports research in distributed search algorithms without the additional features provided in more general toolkits such as PVM [2] or MPI [15], the more interesting parallelization issues are algorithmic ones. What is perhaps most remarkable about NICE is that such a simple and lightweight infrastructure naturally supports parallelization of a broad array of search algorithms.

4. Applications of nagging

While, as we have seen in Section 2, the main idea that underlies nagging is quite simple, there are a number of important details (e.g., the design of appropriate problem transformation functions) that have a direct effect on how well nagging works. These issues are best discussed in the context of specific search algorithms: here, we show how three well-known search algorithms drawn from the artificial intelligence literature—the Davis–Putnam algorithm, the A^* search algorithm, and the $\alpha\beta$ minimax search algorithm—can be parallelized using nagging.

4.1. The Davis–Putnam algorithm

First proposed by Davis and Putnam, and later refined by Loveland, the Davis–Putnam algorithm is the fastest known solution technique for Boolean satisfiability problems that is both sound and complete.[2] We say it is complete because if there is a solution, the Davis–Putnam algorithm guarantees that solution will eventually be found. Other fast solution methods for satisfiability problems such as GSAT, WSAT or simulated annealing are local search procedures that are sound, but not complete [36]. Note that completeness

---

[2] Recall a *Boolean variable* is a variable that can only be *true* or *false*. A *Boolean formula* consists of variables related via the usual logical connectors ¬, ∧, ∨, → and ↔; a formula is in *conjunctive normal form* (CNF) if it is a conjunction of *clauses*, where each clause is a disjunction of *literals*, and each literal is a Boolean variable or its negation. By definition, the SAT-CNF problem (determining whether or not a given CNF Boolean formula is satisfiable) is NP-complete; that is, it is possible to certify a solution is correct in polynomial time, but it is commonly believed that actually finding a solution requires exponential time [5].

does not necessarily imply an exhaustive search; rather, if the given problem is satisfiable, any solution is equally correct, so we can terminate the search as soon as any solution is found. The search space must only be searched exhaustively when we need to guarantee an unsatisfiable problem has no solution. This kind of search-until-first-solution behavior arises most often in automated deduction or theorem proving environments; precisely the contexts where nagging was first proposed.

The Davis–Putnam algorithm solves a particular type of Boolean satisfiability problem, usually called SAT-3-CNF or simply 3SAT, that deals only with Boolean formulas having at most three literals per clause (note that any Boolean formula that can be expressed in CNF can also be expressed in 3CNF by direct manipulation along with the addition of some number of new Boolean variables) [16]. The general idea is simple; if there are a total of N variables, then one can systematically examine all 2^N possible combinations of truth assignments in order to determine if at least one of these truth assignments satisfies the formula.

Two observations serve to reduce the number of variable combinations the Davis–Putnam algorithm need look at in practice. First, if a partial solution having m variable bindings is inconsistent (that is, fails to satisfy one of the clauses), then all $2^{(N-m)}$ completions of this partial solution must also be inconsistent and can be safely pruned: there can exist no solution in this portion of the search space. Second, if a partial solution contains the negations of any $k - 1$ literals from a given k-literal clause, then, if there is to be any hope of finding a satisficing solution, the lone remaining literal in the clause must be satisfied by binding its variable appropriately. Of course, once bound, the newly bound variable may force other variable bindings as well, thus effectively reducing the search space: this process is called *unit propagation*. Thus the Davis–Putnam algorithm operates by systematically examining combinations of truth assignments, with periods of unit propagation occurring whenever possible (see Fig. 2).

What is the expected solution time for this algorithm? In principal, larger formulae define a larger search space and therefore entail longer solution times. In practice, however, exactly how much time is required depends on more subtle characteristics of the specific problem instance (e.g., ratio of number of variables to number of clauses as well as the distribution of variables within the clauses themselves). Put simply, not all like-sized 3SAT problems are equally hard [26,43]. Some problem instances can be easy (imagine, for example, a Boolean formula in CNF where every clause shares a single literal), while others of exactly the same size may result in exponential-time performance. In practice, the order with which variable settings are tried has a critical effect on the time to solution. Numerous *splitting rules*, or heuristics for "good" orderings, exist, but no universal rule will work for all problem instances: existence of a universal splitting rule providing polynomial time performance on all 3SAT problem instances would imply P = NP.

Parallelizing the algorithm of Fig. 2 using partitioning is relatively straightforward: we explore recursive calls to *search*() on separate processors as long as additional processors are available. Of course, we can't really know *a priori* how large each individual subspace is, as some branches may quickly lead to an inconsistent partial solution. This means that it is hard to ensure that the work is fairly distributed over the available processing elements. Furthermore, without some notion of subspace size, it is difficult to determine whether a processor assigned to a subspace has actually failed, gone offline, or is simply taking a

```
3sat(F : formula) : boolean;
    return(search(F, extractVars(F)));

search(F: formula, S: variables) : boolean;
    V: variable ← head(S);
    C: clause;

    if (F = ∅) then return(true);
    elseif (∃C ∈ F | size(C) = 0) then return(false);
    elseif (∀C ∈ F | V ∉ C ∧ ¬V ∉ C) then return(search(F, S\{V}));
    elseif (search(propagate(substitute(F, V)), S\{V})) then return(true);
    elseif (search(propagate(substitute(F, ¬V)), S\{V})) then return(true);
    else return(false);

substitute(F: formula, V : variable) : formula
    G : formula ← ∅;
    C : clause;

    for C in F
        if(V ∉ C)
            G ← G ∪ (C\{¬V});
    return(G);

propagate (F : formula) : formula
    C : clause;

    while(∃C ∈ F | size(C) = 1)
        F ← substitute(F, head(C));
    return(F);
```

Fig. 2. Davis–Putnam algorithm for 3SAT. The problem instance is a Boolean formula F in 3CNF represented as a list of clauses, where each clause is a list of at most three variables, and each variable is either a literal or a negated literal. The *search*() function operates recursively, removing satisfied clauses from F until either there are no more clauses left in F or one of the clauses in F is shown to be unsatisfiable. The splitting rule is encoded explicitly in the ordering and pattern of negations in the list S, which initially contains all of the literals in F, some of which may be negated. Variables whose values are set "early" by unit propagation are "skipped" in the third clause of the large conditional statement. The *substitute*() function constructs and returns G, a new copy of F with satisfied clauses filtered out and references to the negated sense of variable V removed from the remaining clauses. The functions *head*() and *size*() functions return the first element and the cardinality of their argument, respectively, and the function *extractVars*() returns a list of the variables contained in the given formula.

long time to search what turned out to be an unexpectedly large space. Addressing load balancing and fault tolerance issues can only add to the overhead costs associated with partitioning [32].

Compare this partitioning strategy with nagging (see Fig. 3). At initialization, each nagger is provided with the original Boolean formula that specifies the 3SAT problem we wish to solve. A nagging episode begins when the nagger requests a nagpoint from its master, who briefly interrupts its own search to select a random nagpoint. Each nagpoint corresponds to one of the partially-instantiated Boolean formulae considered by the master in the course of its recursive calls to *search*(). Upon receiving the nagpoint, the master resumes its own search, while the nagger first applies a problem transformation function to the nagpoint (e.g., by reordering the list of as-yet-unbound variables, or by randomly

---

$status \equiv true \mid false \mid abort$

nag3sat(F : *formula*) : *boolean*;
  N : *formula*;
  result : *status*;

  niceInit();
  if(**niceRoot**()) **then return**(**searchExplicit**(F, **extractVars**(F)));
  else
    while(*true*)
      $N \leftarrow$ **niceIdle**();
      result $\leftarrow$ **searchExplicit**(N, **transform**(**extractVars**(N)));
      if (*result = true*) **then niceSolve**(N, *result*);
      elseif (*result = false*) **then nicePrune**(N);

Fig. 3. Sketch of nagging implementation for the algorithm of Fig. 2. The call to *niceInit*() connects to the NICE infrastructure and ensures copies of this process are spawned on all participating processors; it also ensures spawned processes are provided with copies of the original problem. The root process (the function *niceRoot*() returns true only on the root processor) performs the normal Davis–Putnam search, while non-root processes engage in a series of nagging episodes, each initiated when *niceIdle*() requests a new nagpoint, denoted N. The function *searchExplicit*() is logically equivalent to the *search*() function of Fig. 2, but with explicit stack manipulation and interrupt handling capabilities for processing messages to/from parent/child processes (convenient primitives to support this functionality are provided in the NICE API). For the nagging processors, the *transform*() function randomly reconfigures the splitting rule as described in the text. Depending on the outcome of the nagger's search, the nagger may pass a solution to its parent (function *niceSolve*()) or force its parent to backtrack (function *nicePrune*()). Note that a nagger's search may also be interrupted by its parent if the parent exhausts the space rooted at the nagger's nagpoint before the nagger does; in this case, *searchExplicit*() would immediately return *abort*, and the nagger would simply request a new nagpoint.

inverting their logical sense, thereby switching the order in which V and $\neg V$ are explored), and then begins its own search. Should the nagger find an assignment that makes the formula true, it interrupts the master and provides the solution, which the master can then in turn provide as the solution to the problem. Should the nagger instead exhaust its space without finding a solution, it can interrupt the master and force the master to backtrack past the nagpoint. Should the master backtrack over the assigned nagpoint before the nagger completes its own search, the master should abort the nagger, who is then free to seek a new nagpoint from the master.

Of course, simply racing the nagger against the master may not produce useful speedups; what is really needed is a good problem transformation function that will increase the nagger's chances of beating the master within the subspace defined by the nagpoint, hence reducing the master's own search space. The insight is that a serial search procedure must necessarily commit to searching a single incarnation of the current problem's search space, while alternate versions of the same search space may entail differing effort to search. Since expected time to solution for a given problem instance is critically dependent on the splitting rule used, changing the splitting rule may lead the nagger to complete its search more quickly than the master.

To gauge if such a strategy might succeed, we can, as a first approximation, empirically examine how a random permutation applied to both variable selection and descendent ordering (i.e., a random splitting rule) affects the overall solution time. We randomly
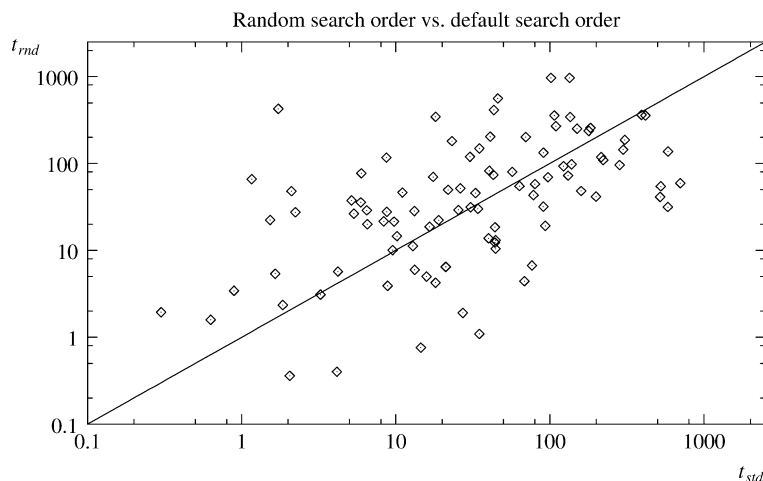
Fig. 4. Davis–Putnam algorithm applied to 100 randomly-generated "difficult" 3SAT problems of various sizes. Each problem is solved twice, once using the default search ordering (t_{std}) and the second time with the permutation transformation applied before solving (t_{rnd}). Results are shown on a log–log scale for clarity. Had all the datapoints been tightly clustered along the diagonal $t_{rnd} = t_{std}$ line, the probability of a nagging processor beating the master would be low; since this is not the case here, permutation appears to be a good candidate problem transformation function for this particular domain.

generate 100 "difficult" 3SAT problems of varying sizes [37]. For this demonstration, time to solution is measured and recorded twice for each 3SAT: once using the default search ordering (t_{std}) and the second time with the random permutation transformation applied before solving (t_{rnd}). The results are shown in Fig. 4; each datapoint in the graph represents the CPU time required to solve the permuted problem (ordinate) plotted against the CPU time for default problem (abscissa). Datapoints appearing below the diagonal $t_{rnd} = t_{std}$ line represent problems that were solved more quickly with the permutation function applied, while those falling above the diagonal were solved more quickly using the default ordering.

We observe that the random splitting rule beats the default splitting rule about half the time, sometimes by a significant margin (note the plot uses a log–log scale for clarity, although unfortunately this somewhat obscures the magnitudes of the differences). The datapoints do not lie tightly clustered about the diagonal line: the farther off the line they are, the greater advantage one might expect to see in a nagging system using this problem transformation function.[3] This demonstration illustrates how speedups are possible even if only a single nagging episode is allowed per problem and even if the selected nagpoint is always the root node of the search. Indeed, since each nagging episode gives you another

---

[3] Of course, this is just an example; more effective problem transformation functions might make use of alternative splitting heuristics, or might even elect to throw away a subset of clauses in order to decrease the size of the search space. In the latter case, solutions in the reduced space no longer correspond to solutions of the original Boolean formula; however, failure to find a solution for this smaller space still implies no possible solution exists for the original space, so the master can still be forced to backtrack.

chance of beating the default splitting rule, multiple nagging episodes over a broad array of properly selected nagpoints should have a better chance of improving a system's overall search performance, an effect we'll observe in the experiments of Section 6.

4.2. A^* search

Consider the well-known *traveling salesperson problem*, or TSP. The problem is as simple to state as it is hard to solve:

Given a collection of N points, find the shortest tour that visits each point and returns to the original starting point.

TSP problems lie hidden in a surprisingly large number problems from operations research and engineering; since TSP is known to be NP-hard, much research has focused on appropriate algorithms and the use of heuristics to find good yet less-than-optimal solutions quickly [33]. Yet devising an algorithm that is guaranteed to provide the (globally) optimal solution is simple, if one is willing to accept poor worst-case performance. Here, we examine one such optimal algorithm for Euclidean TSP (i.e., where the points lie in Euclidean space) based on A^* search, a heuristically-guided branch-and-bound search strategy.[4]

The obvious solution technique is to enumerate all possible tours and then return the shortest one. Starting with an arbitrary 3-point tour and using symmetry to reduce the space, it is easy to see that there are $(N-1)!/2$ different possible tours. We can do a little better by showing that the H points defining the convex hull of the point set must appear in fixed order in any optimal tour; by starting with the convex hull (as opposed to a random 3-point tour), we can reduce the size of the search space, slightly, to $(N-1)!/(H-1)!$. In either case, an exhaustive search algorithm operating on this search space will require, by Sterling's formula, $O(N^N)$ time.

The basic insight required to turn this simple enumeration algorithm into a branch-and-bound algorithm is that information garnered during the search can be used to reduce the combinations that must be examined. If the cost of the current partial solution is greater than that of the shortest solution found so far, we can exclude all completions of the partial solution from the search—since, in Euclidean space, adding more points to the tour can only make it longer—without sacrificing optimality. We can do even better by incorporating a heuristic estimate of the cost to complete a partial tour, pruning subtrees rooted at partial tours where the partial tour cost plus an estimate of the additional cost required to complete the tour exceeds the cost of the current best solution. If the heuristic estimate used always underestimates the true additional cost of the partial solution, we say the heuristic is *admissible*, and it can be shown that the resulting A^* search algorithm will

---

[4] Note that we are not recommending this as a solution technique for TSP problems encountered in practice; rather, we are using TSP as an intuitively accessible example with which to describe the parallelization of A^* search. Most real applications would be better served by using one of the many efficient heuristic algorithms for TSP that yield good, but not optimal, solutions, although advanced cut techniques have been combined with partitioning strategies to find optimal solutions to problems as large as 15,000 points [1].

```
tsp(S : points) : tour
   H : tour ← convexHull(S);
   return(search(H, (S\points(H)), ∅));

search(T : tour, S : points, B : tour) : tour
   P : point;
   if(S = ∅ ∧ (B = ∅ ∨ cost(T) < cost(B))) then return(T);
   elseif(S = ∅) then return(B);
   elseif(cost(T) + estimate(S) ⩾ cost(B)) then return(B);
   else
      P ← head(S);
      for i from 0 to |T| by 1
         B ← search(insert(P, i, T), (S\{P}), B);
      return(B)
```

Fig. 5. A^* algorithm for Euclidean TSP. The *tsp*() function takes a set of points S as input and returns the lowest-cost tour. The heart of the code is the recursive function *search*(), which takes a partial tour T, a set S of points yet to be visited, and B, the lowest-cost tour found so far, and recursively explores the space rooted at that partial tour. Note that if no solution better than B can be found below T, then there is no need to search any further. The *cost*() function returns the cost of its tour argument, or 0 if the argument is the null tour ∅, while *estimate*() returns a lower bound on the additional cost of adding the points in its argument to any existing partial tour. As before, the function *head*() returns the first element of its argument, while *points*() returns the point set of its tour argument and *insert*() inserts a new point P into the ith position of partial tour T. Parallelization of this algorithm with nagging follows in a manner similar to the sketch given in Fig. 3.

still return the optimal solution while exploring no more nodes (and generally far fewer nodes) than branch-and-bound search (see Fig. 5) [28].

As for the Davis–Putnam algorithm, applying partitioning to the serial algorithm of Fig. 5 is relatively simple: the idea is to explore different recursive calls to *search*() on separate processors [4,23,31]. Of course, load balancing and fault tolerance issues are present here, just as with the Davis–Putnam algorithm. But there are also additional complications due to fact that A^* search is not just a *satisficing search* (i.e., search to first solution) but rather an *optimizing search* (i.e., search to best solution). The difference is that every node in the search space must be either searched or safely pruned so that the search procedure can certify that no better solution exists (compare with the Davis–Putnam algorithm, where only unsatisfiable formulae entail an exhaustive search, while finding any solution to a satisfiable formula terminates the search immediately). The net effect is to make the subspaces more interdependent: to see why this is so, consider what happens when a new (and presumably favorable) best solution is found in the first subspace explored by a serial search process. Clearly, the new solution's lower cost may lead to significantly less search in subsequent subspaces. When these subspaces are searched in parallel, however, this new best solution may not be discovered until most of the work in the other subspaces has already been performed. And even if the new best solution is found soon enough to have an impact on the concurrent subspace searches, its cost must be communicated to all the other processors, entailing some additional communication overhead.

We now consider parallelizing the serial algorithm of Fig. 5 using nagging; the general idea is identical to that of Fig. 3 for the Davis–Putnam algorithm. Idle naggers request a nagpoint (corresponding to a partial tour passed in one of the recursive calls of the *search*() function still on the master processor's calling stack) and search this space in parallel. Unlike the partitioning case, there are no load balancing or fault tolerance problems here: nagpoints are generated whenever they are solicited, and lost or unresponsive naggers can't affect the master's own search. Of course, as with the Davis–Putnam algorithm, the key to effective nagging—that is, nagging that actually provides some speedup—is an effective problem transformation function. One reasonable approach might be to randomly perturb the order in which points are inserted into the growing partial tour (e.g., by reordering the list of as-yet-unvisited points), or perturb the order in which descendent nodes are generated (e.g., by changing the insertion order within the loop of the *searchExplicit*() function). Alternatively, we might rely on problem-specific knowledge and choose a mixture of the many TSP-specific heuristic ordering strategies from the operations research literature. Or one might even adopt an *abstraction transformation*, where a nagger simply throws away a certain number of points in the point set. This transformation can still provide speedup if the optimal tour in the abstracted search space is longer than the master's current best solution (in Euclidean space, the cost of the optimal tour for the reduced problem is a lower bound to the cost of the optimal tour on the original points). We'll look at how nagging compares with partitioning for the Euclidean TSP algorithm just described in Section 6.

4.3. SPAM: $\alpha\beta$ minimax search

Historically, much research within the artificial intelligence community has focused on the problem of playing zero-sum two-player games, such as chess or checkers. Most of this research involves derivatives or variants of $\alpha\beta$ minimax search, which is itself a straightforward refinement of the original notion of minimax search. Within this field, various forms of parallelism have also received a lot of attention [3,8,9,17]. Here, we show how the $\alpha\beta$ minimax search algorithm can be parallelized using nagging, producing an algorithm we call *SPAM*, for *Scalable Parallel Alpha-Beta Minimax*.

The idea underlying any minimax procedure is to generate the tree of legal moves to a fixed depth (given by the *ply* argument) and evaluate the quality of the resulting board positions using a *static board evaluation function*, or *SBE*.[5] The SBE looks at a board and returns a value on $[-\infty, \infty]$, with $-\infty$ representing a loss and ∞ representing a win for the specified player. By selecting the maximum/minimum values at alternating levels, these leaf SBE values can be propagated up to the root of the tree, by definition a maximizing level, where they can be used to select the branch leading to the best attainable outcome for the current player. Deeper searches generally lead to more informed choices; unfortunately,

---

[5] There exist numerous alternative formulations of the minimax search procedure. This particular formulation was selected for its simplicity. Of course, any formulation of the minimax algorithm assumes that what is good for one's opponent is symmetrically bad for the player him or herself, and that the opponent is a rational one, making decisions based on an identical SBE. Also, note that our formulation only returns the best SBE value found in the tree of given ply rooted at the given board. In practice, the procedure should also return the corresponding move.

```
alphabeta(B : board, P : player, D : integer) : score
   return(search(B, P, D, −∞, +∞)

search(B : board, P : player, D : integer, α : score, β : score) : score
   M : move;
   if(won(B, opponent(P))) then return(α);
   elseif(D = 0) then return(SBE(B, P));
   else
      for M in legalMoves(B, P)
         α = max(α, −search(applyMove(B, M), opponent(P), D − 1, −β, −α);
         if(α > β) return(α);
      return(α);
```

Fig. 6. "Negamax" formulation of $\alpha\beta$ minimax search algorithm. Return the best possible score player P can hope to attain in the search space of D depth rooted at board B. Pruning occurs when α exceeds β within the loop over the possible legal moves at this board position, causing the function to return immediately. We assume the *won*(), *legalMoves*(), *SBE*(), and *applyMove*() functions have the appropriate game-specific definitions; *opponent*() returns the other player, and *max*() has the usual semantics.

the size of the game tree that must be examined increases exponentially with the depth of exploration. Thus for meaningfully large games (e.g., chess) this exponential growth makes all but the most shallow searches impractical.

The intuition underlying $\alpha\beta$ *pruning* is to exploit information generated during the exploration of one portion of the search space to justify skipping or pruning other parts of the space. Judicious pruning can extend the computational horizon under which the search operates, allowing deeper searches in the same amount of computation time. Of course, one can't really beat the exponential nature of the search: there will always be some practical maximum on the depth of search. The idea is to extend that maximum as much as possible by reducing the number of branches that must be examined.

Extending the minimax algorithm to perform $\alpha\beta$ pruning is relatively straightforward; the basic idea is to pass two additional parameters called α and β, which serve as bounds on the interesting values at any given node (see Fig. 6). Paths whose leaf values are guaranteed not to fall in the interval $[\alpha, \beta]$ (initially $[-\infty, +\infty]$) would never be chosen by the minimax procedure and therefore can be safely ignored. As the game tree is searched, α values will only increase and β values will only decrease, further constraining the search. It should also be clear that, by construction, $\alpha\beta$ minimax search produces identical choices to minimax search in all cases. Furthermore, as with A^* search, the search reduction produced by $\alpha\beta$ pruning depends on the order in which paths are searched; for some pathological game trees, $\alpha\beta$ minimax and standard minimax will search identical game trees.[6] From an analytic perspective, the number of calls to the SBE will vary between roughly $2b^{d/2}$ (the best case) and b^d (the worst case) for a game tree of depth d with uniform branching

---

[6] Indeed, many real-world implementations of $\alpha\beta$ minimax search try to improve performance by generating internal choice points in an ordered fashion, usually guided by a cheap, fast, secondary SBE function applied to the board positions represented by these internal nodes. Of course, board positions which look "bad" locally may actually turn out to be "good" several ply deeper, so internal choice point reordering is nothing more than a greedy optimization technique that may lead to—but does not guarantee—more efficient search.

factor b [18]. Thus while the exponential factor is still present, it may be significantly reduced, allowing deeper, more informed, searches in the same amount of time.

While implementing $\alpha\beta$ minimax search requires only minor extensions to the basic minimax procedure, these modifications make attempts to parallelize the search significantly more complicated. This is because minimax search is easily decomposed at any node, and game trees rooted at each child node can be considered independently by separate processing elements. Once child SBE values are computed in parallel, it is a simple matter to compare these estimates and select the best alternative. In contrast, for $\alpha\beta$ minimax search, partitioning and parallel exploration of the game tree may well require more time than the corresponding serial search. This is because the savings realized by $\alpha\beta$ pruning over standard minimax search result from exploiting the information obtained searching one part of the game tree while exploring another (like for A^* search, but unlike for the Davis–Putnam algorithm). Even if one could instantly share new α and β values with all of the processing elements at zero communication cost, the new values may come too late to matter. In short, the parallel algorithm might well search a greater number of nodes than would the serial algorithm.

The SPAM algorithm exploits all of the search constraints embodied by the α and β values while keeping communication between processing elements infrequent and brief. The nagging algorithm (see Fig. 7) is essentially identical to that described for A^* search

---

$nagpoint = \{B : board, \; P : player, \; D : integer, \; \alpha : score, \; \beta : score\}$

nagalphabeta($B : board, \; P : player, \; D : integer$) $: score$
   $N : nagpoint;$
   $result, \alpha', \beta' : score;$

   niceInit();
   if (**niceRoot**()) **then return** (**searchExplicit**($B, P, D, -\infty, +\infty$)
   else
     while(*true*)
       $N \leftarrow$ **niceIdle**();
       $\{\alpha', \beta'\} \leftarrow$ **narrow**($N.\alpha, N.\beta$)
       $result \leftarrow$ **searchExplicit**($N.B, N.P, N.D, \alpha', \beta'$)
       if ($result > \alpha' \wedge result < \beta'$) **then niceSolve**($N, result$);
       elseif ($result = \alpha'$) **then niceRestrict**($N, \; N.\alpha, \; result$);
       elseif ($result = \beta'$) **then niceRestrict**($N, \; result, \; N.\beta$);

Fig. 7. Sketch of nagging implementation for the algorithm of Fig. 6. As for Fig. 3, the function *searchExplicit*() is identical to the *search*() function of Fig. 6, but with explicit stack manipulation and interrupt handling capabilities. The root process performs the normal $\alpha\beta$ minimax search, while non-root processes engage in a series of nagging episodes, each initiated when *niceIdle*() requests a new nagpoint. Nagpoints are again denoted N, but here consist of a board position, the next player to move, a search depth limit, and α and β values. For the nagging processors, the *narrow*() function randomly reduces the search range as described in the text, and serves as a part of the problem transformation function in combination with permutations, provided by modifying the nagger's copy of the *legalMoves*() function (Fig. 6) to randomly perturb the sequence of legal moves generated. Depending on the outcome of the nagger's search, the nagger may pass a solution to its parent (function *niceSolve*()) or force its parent to refine its own α or β parameters (function *niceRestrict*()). Note that a nagger's search may also be interrupted by its parent if the parent exhausts the space rooted at the nagger's nagpoint before the nagger does; in this case, *searchExplicit*() would immediately return *abort*, and the nagger would request a new nagpoint.

and the Davis–Putnam algorithm, except that here we will exploit a problem transformation function based on both permutation and *window narrowing*. The idea is that a nagger can artificially restrict its master's $\alpha\beta$ window; using a narrow interval $[\alpha', \beta']$ (where $\alpha < \alpha'$ and $\beta' < \beta$) ensures that the nagger's search procedure will prune aggressively, exploring only a relatively small number of nodes in the search space, thereby increasing the odds that it will beat the master (of course, we can also permute the node exploration order as well).[7] In exchange for the reduction in search, the value computed by the nagger's transformed search may not always be directly substituted for the true value that would be computed by the master for the same subspace. More precisely, if the nagger returns a value that is greater than α' and less than β', then this corresponds to the true value that would be computed by the master, and we can force the master to backtrack and continue its search from that point on. But if the nagger returns a value less than α' (alternatively: greater than β'), then it means that the true value lies between α and α' (alternatively: β' and β). This information can be used to reduce the master's search space by setting the master's β to α' (alternatively: α to β'), which may well lead to immediate improvement of the search efficiency for the master's current search. In Section 6 we'll empirically examine the behavior of SPAM with this problem transformation function.

5. Analysis

We now introduce simple analytical models of both nagging and partitioning that help to explain how and when nagging can be expected to provide a performance advantage over more traditional partitioning methods. Our analysis relies on techniques developed for reliability data analysis; specifically, understanding how a problem transformation function used defines a probability distribution of solution times for a specific problem [21,24,25].

Let us assume that a random variable x drawn from a distribution X represents the solution time of a specified problem under a given problem transformation function. Of course, the exact nature of X depends on the search algorithm and problem transformation function applied; we'll examine different choices for X later. The behavior of x can be described by its density function $f(x)$ and its corresponding cumulative density function $F(t)$, which measures the probability that the solution time x is less than some specified time value t:

$$F(t) = Pr(x \leqslant t) = \int_{x=0}^{t} f(x)\,dx.$$
(5.1)

Note that the physical interpretation of x imposes certain constraints on allowable values for $f(x)$ and $F(t)$: more precisely, $f(x) \geqslant 0$, $f(x) = 0$ for $x \leqslant 0$, $F(0) = 0$, and $F(\infty) = 1$ follow from the fact that real problems are never solved in less than zero time and the probabilistic semantics of $F(t)$.

We can use this statistical model to study the behavior of the coarsest possible form of nagging, where each of the n nagging processors operates on an identical copy of the entire

---

[7] In the limit, where $\alpha' + \varepsilon = \beta'$ this reduces to zero-window search [29].

problem instance. In essence, the n naggers are racing to find a solution, each operating on a different, solution-equivalent, transformation of the original space.[8] Once any processor completes its search, the solution it finds (or fails to find) applies without modification to the original problem instance. In practice, multiple naggers usually search different, randomly selected, nodes along the master's current search path in service of a master search process; furthermore, some of the more effective transformation functions in use may not be solution equivalent (see, e.g., the abstraction transformation of Sections 4.1 and 4.2, and the window narrowing transformation of Section 4.3).

Let the random variable v_n represent the time elapsed before one of the n independent processors finds the problem solution. Clearly, for the coarse nagging model, $v_n = \min(x_1, x_2, \ldots, x_n)$ where each x_i is an independent random variable drawn from the original distribution X. Let $G_n(t)$ be the corresponding cumulative density function of v_n. Since $v_n = \min(x_1, x_2, \ldots, x_n)$, and each x_i is independent, $G_n(t)$ represents the probability that at least one of the x_i values is less than t, which is 1 minus the probability that all the x_i values exceed t. Thus using Eq. (5.1):

$$G_n(t) = Pr(v_n \leqslant t) = 1 - \prod_{i=1}^{n}\big(1 - F(t)\big) = 1 - \big(1 - F(t)\big)^n. \tag{5.2}$$

Taking the derivative of $G_n(t)$ with respect to t evaluated at v_n yields the density function $g_n(v_n)$:

$$g_n(v_n) = nf(v_n)\big(1 - F(v_n)\big)^{n-1}. \tag{5.3}$$

A similar argument can be made to construct a coarse model of partitioning. Technically, the argument is somewhat more problematic, since once a problem is partitioned and distributed to different processors, each processor is indeed solving a different problem, whose solution time distributions might vary significantly from the original one. However, as is the case with nagging, we can make reasonable assumptions to support some crude—but still informative—comparisons between the two models.

First, we assume that the original problem, whose solution time is still described by a random variable x drawn from a distribution X, is partitioned in n subproblems each having identically-sized search spaces; essentially, we're claiming a perfect *a priori* solution to the load balancing problem. Second, we assume that fault tolerance is not an issue, and that all processors actually will terminate their search and return their partial solutions. Third, we assume that run times scale linearly, that is, that the run time of a subproblem of size $1/n$ is governed by x/n, where the random variable x refers to the solution time of the original problem. Finally, we assume that the cost of merging the subproblem solutions together to form a solution to the original problem is negligible. The last assumption is the most problematic, since for NP-hard problems the cost is likely to be high if the merger is even feasible. However, since it is our intent to compare this model

---

[8] A *solution-equivalent transformation* is one that transforms the original space without losing any existing solutions or adding spurious solutions; the permutation transformation is a good example of a solution-equivalent transformation, while the abstraction transformation of Section 4.2 and the window narrowing transformation of Section 4.3 are not.

of partitioning with nagging, we can afford to make generous assumptions on behalf of partitioning without compromising the essence of our comparisons.

With these assumptions in place, the time required to solve the partitioned problem under this coarse partitioning model is described by a random variable w_n defined as $w_n = \max(x_1/n, x_2/n, \ldots, x_n/n)$, since it is necessary to solve each of the subproblems before merging their solutions together.[9] Proceeding in the same fashion as for nagging, we obtain the density function $h_n(w_n)$ from the cumulative density function $H_n(t)$ as follows. Since $H_n(t)$ represents the probability that $w_n < t$, i.e., all the x/n are less than t, we have

$$H_n(t) = Pr(w_n \leqslant t) = \prod_{i=1}^{n} Pr\left(\frac{x_i}{n} < t\right) = F(nt)^n \qquad (5.4)$$

from which we obtain

$$h_n(w_n) = n^2 f(nw_n)\big(F(nw_n)\big)^{n-1}. \qquad (5.5)$$

Note that $g_1(v_1) = h_1(w_1) = f(x)$ and thus $G_1(t) = H_1(t) = F(t)$, just as one should expect given that the single processor system is the trivial case of both the coarse nagging and coarse partitioning models.

Once the appropriate distribution X has been fixed, it is relatively easy to make performance comparisons between serial execution, coarse nagging and coarse partitioning by comparing their expected solution times.

5.1. The uniform distribution

The first sample distribution we look at is the simplistic case where X is the uniform distribution with values x ranging between two constants t_{lo} and t_{hi}.[10] For this uniform distribution, the density function is

$$f(x) = \frac{1}{(t_{hi} - t_{lo})} \qquad (5.6)$$

for $t_{lo} \leqslant x \leqslant t_{hi}$ and $f(x) = 0$ elsewhere. It easy to see that

$$F(t) = \int_{x=t_{lo}}^{t} \frac{1}{(t_{hi} - t_{lo})} \, \mathrm{d}x = \frac{t - t_{lo}}{t_{hi} - t_{lo}} \qquad (5.7)$$

---

[9] Unfortunately, the model becomes more complicated for satisficing—as opposed to optimizing—search. Consider the Davis–Putnam algorithm: if the Boolean formula is true, the algorithm will terminate when the first subproblem that finds a solution terminates, or $w_n = \min(x_1/n, x_2/n, \ldots, x_n/n)$. On the other hand, if the Boolean formula is false, the search will have to exhaust the entire search space to guarantee no solution is overlooked, and thus the required time should be $w_n = \max(x_1/n, x_2/n, \ldots, x_n/n)$ as given in the text. Thus a mixture model that blends these two cases together might provide a more appropriate model of satisficing search.

[10] From a practical perspective, the uniform distribution is not of great interest, since there is relatively little reason to believe solutions times of real problems would fit. Nonetheless, it does serve as useful point of comparison for the other distributions considered later in this paper.

for $t_{lo} < t < t_{hi}$, $F(t) = 0$ for $t \leqslant t_{lo}$ and $F(t) = 1$ for $t \geqslant t_{hi}$. Applying Eq. (5.3), we obtain the density function for an n-processor coarse nagging system:

$$g_n(v_n) = \frac{n(t_{hi} - v_n)^{n-1}}{(t_{hi} - t_{lo})^n}. \tag{5.8}$$

In a similar fashion, but using Eq. (5.5) for the n-processor coarse partitioning model, we obtain

$$h_n(w_n) = n^2 \frac{(nw_n - t_{lo})^{n-1}}{(t_{hi} - t_{lo})^n}. \tag{5.9}$$

5.2. The exponential distribution

Using the same approach we can consider other, more realistic, probability distributions. Here we look at an exponential distribution with a fixed minimum time t_{lo} and decay parameter λ. The exponential distribution has long been used to model equipment failure in reliability studies; it follows from a uniform random failure pattern, modeled as a Poisson process [24]. This distribution's density function is given by

$$f(x) = \lambda \frac{e^{-\lambda x}}{e^{-\lambda t_{lo}}} = \lambda e^{\lambda(t_{lo} - x)} \tag{5.10}$$

with cumulative density function

$$F(t) = -e^{\lambda(t_{lo} - t)}. \tag{5.11}$$

Appropriate substitution in Eq. (5.3) yields

$$g_n(v_n) = n\lambda e^{\lambda n(t_{lo} - v_n)} \tag{5.12}$$

for the coarse nagging case, while Eq. (5.5) produces

$$h_n(w_n) = n^2 \lambda e^{\lambda(t_{lo} - nw_n)} \left(1 - e^{\lambda(t_{lo} - nw_n)}\right)^{n-1} \tag{5.13}$$

for the coarse partitioning case.

5.3. The lognormal distribution

Recently, some have characterized the observed behavior of backtracking search on satisfiability problems using distributions of the Pareto–Lévy form. Such distributions differ from the exponential distribution used in Section 5.3 because they are *heavy tailed*, that is, their complementary cumulative density function $1 - F(t)$ decays slower than exponentially. Heavy tailed distributions have been used to justify a random restart strategy (a sort of single processor version of coarse nagging) for satisfiability problems [12]. Many different heavy-tailed distributions are used in reliability analysis, although the most commonly used are the Weibull and the lognormal distributions (others include the Gumbel or extreme value distribution, the Birnbaum–Saunders distribution, etc.). The key question remains how to choose which distribution best models the observed search behavior—not only for satisfiability problems, but for all the search problems studied here. Fortunately, exploratory data analysis techniques for testing distributional adequacy

such as the Kolmogorov–Smirnov or the (somewhat more sensitive) Anderson–Darling goodness-of-fit tests are well known [38], and support our use of the lognormal distribution for this analysis.[11]

Consider a lognormal distribution with fixed minimum time t_{lo}, scale parameter μ, and shape parameter σ. The distribution's density function is

$$f(x) = \frac{1}{\sigma(x - t_{lo})\sqrt{2\pi}} e^{\frac{-(\log(x - t_{lo}) - \mu)^2}{2\sigma^2}}. \tag{5.14}$$

The cumulative density function can be expressed in terms of Φ, the cumulative density function of the standard normal distribution, or erf, the error function

$$F(t) = \Phi\left(\frac{\log(t - t_{lo}) - \mu}{\sigma}\right) = \frac{1}{2} + \frac{1}{2}\operatorname{erf}\left(\frac{\log(t - t_{lo}) - \mu}{\sqrt{2}\sigma}\right). \tag{5.15}$$

Appropriate substitutions can be made into Eqs. (5.3) and (5.4) to obtain $g_n(v_n)$ and $h_n(w_n)$, although the resulting expressions are not terribly informative since expressions containing erf are notoriously difficult to simplify.

5.4. Comparing nagging and partitioning

We are now ready to make direct comparisons between performance estimates for coarse nagging and for coarse partitioning for some range $1 \ldots n$ of processing elements. One simple comparison is to look at the *performance ratios*, defined as the ratio of the serial expected solution time $E(x)$ to the parallel expected solution time $E(v_n)$ (for nagging) or $E(w_n)$ (for partitioning), where the expected solution times $E(x)$, $E(v_n)$ and $E(w_n)$ are simply the average elapsed times.

A more meaningful statistic is the *expected speedup*, defined as the expected value of x/v_n for nagging (alternatively: x/w_n for partitioning). We say this metric is more meaningful because it represents the expected speedup observed in an experiment where serial performance is compared directly with parallel performance on each individual problem. We define a new random variable $\phi_n = x/v_n$ (alternatively: $\psi_n = x/w_n$) and compute its expected value $E(\phi_n)$ (alternatively: $E(\psi_n)$). Since both ϕ_n and ψ_n are ratios, we consider their geometric means, that is

$$E^*(\phi_n) = e^{E(\log(\phi_n))} \tag{5.16}$$

---

[11] We generated four sets of 100 datapoints each by solving a single problem for each of A^*/TSP, Davis–Putnam/3SAT/unsatisfiable, Davis–Putnam/3SAT/satisfiable, and SPAM/Othello 100 times using a strictly solution-equivalent problem-transformation function (i.e., permutation in this case). We then applied the Anderson–Darling test to see which of the set of tested distributions (normal, lognormal, exponential, Weibull, Gumbel, and logistic) were consistent with the observed data. In all but two cases, the Anderson–Darling test rejected ($p < 0.05$) all of the tested distributions *except* for the lognormal distribution (the exceptions are the Davis–Putnam/3SAT/satisfiable data, where the Anderson–Darling test rejected all but the lognormal and Weibull distributions, and the SPAM/Othello data, where the Anderson–Darling test rejected all but the lognormal and Gumbel distributions). While these tests are not entirely conclusive (they are, after all, based on just a few randomly-generated problems), they do seem to suggest that the lognormal is well suited to modeling the range of search behaviors studied in this analysis.

and

$$E^*(\psi_n) = e^{E(\log(\psi_n))} \tag{5.17}$$

rather than $E(\phi_n)$ or $E(\psi_n)$ directly, since the geometric mean is more representative of the expected speedup over many such trials. Exploiting the additive properties of expected values and the fact that x and v_n (alternatively: x and w_n) are independent, we obtain:

$$E\big(\log(\phi_n)\big) = E\big(\log(x)\big) - E\big(\log(v_n)\big)$$
$$= \int_{x=t_{lo}}^{t_{hi}} \log(x) f(x)\,\mathrm{d}x - \int_{v_n=t_{lo}}^{t_{hi}} \log(v_n) g_n(v_n)\,\mathrm{d}v_n \tag{5.18}$$

and, similarly,

$$E\big(\log(\psi_n)\big) = \int_{x=t_{lo}}^{t_{hi}} \log(x) f(x)\,\mathrm{d}x - \int_{w_n=t_{lo}/n}^{t_{hi}} \log(w_n) h_n(w_n)\,\mathrm{d}w_n. \tag{5.19}$$

Unfortunately, the formulae for $E^*(\phi_n)$ and $E^*(\psi_n)$ are often quite complex in the general case. However, values for both $E^*(\phi_n)$ and $E^*(\psi_n)$ are easily tabulated for specific values of n, which lend themselves to graphical comparison (see Fig. 8). Qualitatively speaking, Fig. 8 makes clear the noticeable scaling advantage of nagging over partitioning within this analytical model, especially for the exponential and log-normal distributions, which correspond more closely to distributions observed in practice.

Another interesting metric is suggested by closer examination of the performance ratio for coarse nagging in the exponential distribution case:

$$\frac{E(x)}{E(v_n)} = \frac{t_{lo} + \frac{1}{\lambda}}{t_{lo} + \frac{1}{n\lambda}}. \tag{5.20}$$

We note that, when both λ and t_{lo} are small, the performance advantage obtained by coarse nagging can—on average—approach n, or linear speedup. This implies that coarse nagging can be expected to provide superlinear speedups with respect to a serial search about half the time in the exponential distribution case. For heavy-tailed distributions, the advantage of coarse nagging is even more decisive, providing some theoretical justification for the observed effectiveness of random restart strategies on serial processors.

More formally, it is interesting to compute and compare the probability that a coarse nagging or partitioning system will exhibit superlinear speedup with respect to the average sequential case, which is easily expressed as:

$$Pr\left(v_n \leqslant \frac{E(x)}{n}\right) = G_n\left(\frac{E(x)}{n}\right) \tag{5.21}$$

for nagging, and

$$Pr\left(w_n \leqslant \frac{E(x)}{n}\right) = H_n\left(\frac{E(x)}{n}\right) \tag{5.22}$$

Speedup vs. Number of Processing Elements (Uniform Distribution)

Speedup vs. Number of Processing Elements (Exponential Distribution)

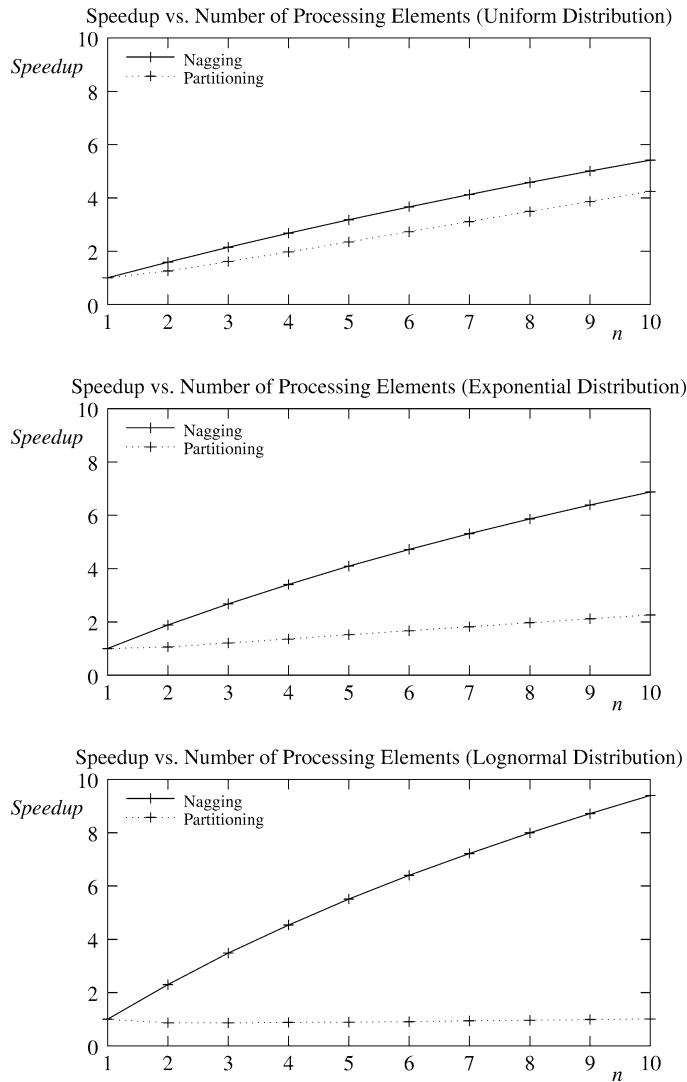Speedup vs. Number of Processing Elements (Lognormal Distribution)

Fig. 8. Expected speedups $E^*(\phi_n)$ and $E^*(\psi_n)$ vs. number of processing elements for uniform ($t_{lo} = 0.01$, $t_{hi} = 1.0$), exponential ($t_{lo} = 0.01$, $\lambda = 2$), and lognormal distributions ($t_{lo} = 0.01$, $\mu = 0.5$, $\sigma = 1.5$). This statistic illustrates the scaling advantage of nagging over partitioning within this simple analytic model for all three distributions studied, and how the advantage of nagging grows as the distribution becomes more heavy tailed.

for partitioning. As before, while the resulting expressions may be difficult to simplify, it is easy to tabulate values for $n = 2, \ldots, 10$ (see Table 1).

It is clear that, for all of the distribution models studied here, the coarse model of nagging retains its potential for producing superlinear speedup as the number of processors

Table 1

Technique	Distribution	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$
Nagging	Uniform	0.43	0.41	0.39	0.38	0.37	0.36	0.36	0.35	0.34
Partitioning	Uniform	0.25	0.13	0.06	0.03	0.02	0.01	0.00	0.00	0.00
Nagging	Exponential	0.62	0.62	0.61	0.60	0.59	0.59	0.58	0.57	0.56
Partitioning	Exponential	0.40	0.25	0.16	0.10	0.06	0.04	0.03	0.02	0.01
Nagging	Lognormal	0.85	0.88	0.89	0.90	0.91	0.91	0.91	0.91	0.91
Partitioning	Lognormal	0.60	0.46	0.36	0.28	0.21	0.17	0.13	0.10	0.08

is increased, while the already limited potential of partitioning to do so rapidly vanishes as more processors are added.

Of course, the analytic models presented here are relatively simple, and do not correspond exactly with how nagging and partitioning are actually applied in practice. We're mostly interested in how well the observations made using the coarse models hold up in more realistic situations: for example, we might nag or partition recursively, use problem distribution functions that are not strictly solution equivalent (e.g., window narrowing or abstraction), or elect to nag or partition multiple times per problem at internal nodes of the search tree rather than just once at the root node. We turn to these rather more practical questions in the next section, using experimentally-obtained quantitative data to support our claims about nagging's performance in a principled manner.

6. Empirical evaluation

Empirical studies, if carefully done, can give a realistic picture of a system's behavior. Here, we focus on performance issues, using experimental data to contrast the relative performance of nagging and partitioning as well as to support our claims regarding the scalability of nagging.

6.1. Experiment 1

The first experiment compares implementations of nagging and partitioning that are purposefully designed to evoke the coarse analytic models of the previous section. The experimental procedure is straightforward. First, for each of the three tested algorithms (Davis–Putnam/3SAT, A^*/TSP, and SPAM/Othello), 100 randomly-generated problems are solved serially using a fixed search order on a 450 MHz Celeron machine running the Linux operating system.[12] Next, a second 450 MHz Celeron machine is added to the NICE hierarchy, and each problem is solved twice more, once using the second machine

---

[12] The random problem sets were generated to provide a good cross-section of solution times ranging from 0.01 seconds (the resolution of the Linux system clock) to roughly 20 minutes on a single processor system. Davis–Putnam/3SAT problems ranged from 120 to 140 variables (with between 514 to 604 clauses), and were, as mentioned in Section 4.1, intended to be "difficult" problems, while the randomly-generated A^*/TSP problems ranged from 29 to 33 cities. The SPAM/Othello problems consisted of random, legal, mid-game Othello boards (having 18 to 22 pieces placed) searched to an 8 or 9 ply horizon.

as a nagging processor and once using is as a partitioning processor instead. For each problem, the solution found and elapsed processor time used by the master processor (as returned by the ANSI C *clock*() function) are recorded. Any search that is not completed by a prespecified resource limit is marked as *censored*, and the best solution found so far is returned for comparison.

Like the coarse analytic models of the previous section, the only problem transformation function used here is the permutation transformation, which is used for all three algorithms (we'll examine other transformation functions in Section 6.2). Unlike the coarse analytic models, however, for optimizing search algorithms (SPAM/Othello and A^*/TSP), more than one nagging event may occur in the course of the experiment. While nagging always takes place at the root node, a nagger that finds a better solution is allowed to immediately report its new bound to the master and begin a new nagging event, applying the newly obtained bound and a new permutation transformation to the root node once more.

A similar change is made for the partitioning case: to address the load-balancing issues normally associated with partitioning, we use an asynchronous partitioning protocol that is similar to that used for nagging. As with nagging, an idle slave processor initiates the process by requesting additional work from its master. Instead of a nagpoint, however, the master provides its slave the last available sibling node of the "highest" available ancestor node along its own search path. The slave then searches this partition using the identical search ordering as the master. When the slave completes its search, it reports the result to the master who then marks the assigned sibling node as solved. The master is then free to assign another partition to the slave. Note that if the master's search enters the subspace assigned to a slave, the slave in effect becomes a nagger, albeit one without the benefit of a problem transformation function, but with a head start on its search space.

All of the 3SAT problems were solved by every system configuration within the prespecified resource limit. For the Othello problems, one of the 100 random problems was censored (not solved within the resource limit) by the serial search system, yet was easily solved by both the nagging and partitioning systems. The situation is more complicated for the TSP problems, as a total of 14 problems were censored by the serial system. Of these 14 problems, five were solved to optimality by both the nagging and partitioning system, and one additional problem was solved to optimality by the nagging system alone. It is important to note that where censoring occurred in both serial and parallel configurations, qualitatively better solutions were produced by the parallel systems (for six of eight such problems, nagging found the shortest tour, while partitioning found the shortest tour in the remaining two cases). So, at least in terms of number of problems solved to optimality as well as quality of solution when problems were not solved optimally, the performance edge appears to belong to nagging.

Differences in solution quality aside, we are mostly interested in quantifying changes in system performance. Here, we compare computed speedup values (recall speedup is defined as the ratio of serial solution time to parallel solution time), where a speedup value of 1.0 implies no difference between the serial and parallel systems, while larger

Table 2

	Nagging speedup					Partitioning speedup				
	N	min	μ	μ^*	max	N	min	μ	μ^*	max
A^*/TSP	92	0.86	2.43	1.51	69.73	91	0.55	1.80	1.68	5.23
SPAM/Othello	100	0.98	3.28	1.55	148.32	100	1.01	2.14	1.75	22.55
Davis–Putnam/3SAT	100	0.45	7.54	2.00	221.00	100	0.55	7.39	1.87	481.17
satisfiable	49	0.45	12.93	2.50	221.00	49	0.55	13.43	2.25	481.17
unsatisfiable	51	0.98	2.36	1.62	14.19	51	1.07	1.59	1.57	1.99

speedups imply the parallel system is faster.[13] Table 2 presents minimum, arithmetic
mean (μ), geometric mean (μ^*), and maximum speedups computed excluding doubly-
censored datapoints for all tested systems. Given the methodological difficulties just noted,
some interpretation is in order. Consider, for example, the A^*/TSP and SPAM/Othello
values shown in the table. For both of these algorithms, the mean speedup μ reported by
nagging is larger than that for partitioning, but the geometric mean μ^* is less than that for
partitioning. This is a consequence of the nondeterministic nature of nagging; the amount
of speedup you get can vary dramatically from one trial to the next, even when solving the
same problem. In contrast, partitioning is by nature more conservative and likely to provide
more uniform amounts of speedup on subsequent trials. This behavior is also at least
partially evident in the maximum speedup values shown, as nagging's best performance
in the test suite represents an order of magnitude improvement over partitioning. Note
also that, as one would expect, the minimum values hover by and large at or just below
1.0. Values below 1.0 represent a performance penalty incurred by the parallel systems
with regard to the serial system. This is partly due to initial setup costs (e.g., connecting
with the NICE infrastructure) and partly due to communication overhead (as we shall soon
see, the smallest values observed are usually associated with problems that can be solved
quickly with a single processor, hence precluding the amortization of startup costs over
longer solution times).

The results for the Davis–Putnam/3SAT problems are notably different from those for
the other tested systems: over the entire set of problems, nagging's measured speedups
exceed those of partitioning (as measured by both μ and μ^*). If restricted to satisfiable
formulae only, partitioning's measured performance is similar to that of nagging. Recall

---

[13] Methodologically, direct comparison of sets of speedup values is somewhat difficult for a number of reasons.
First, as noted earlier, reporting arithmetic means for ratios like speedup is problematic; reporting geometric
means would perhaps be a better choice, but this is not consistent with the general practice of the parallel
processing community, where arithmetic means are the norm. Furthermore, it is important to keep in mind that
the distribution of observed speedup values are quite skewed (not surprisingly, given that, by definition, they are
bounded below at 0): simply reporting summary statistics that evoke normal distributions in the minds of some
readers is misrepresentative. Finally, some caution must be exercised when comparing censored datapoints [34].
Since identical resource limits are imposed on both parallel and serial trials, doubly-censored datapoints will have
unit speedup values. Singly-censored datapoints are harder; fortunately, in our experiments, all singly-censored
datapoints are censored by the serial system (never by the parallel system), so their computed speedup values
represent underestimates of true speedup. Note, however, that direct comparisons of computed speedup values
between uncensored and singly-censored datapoints or singly-censored and doubly-censored datapoints should
be made only with care.

that the Davis–Putnam/3SAT algorithm searches until it encounters a satisficing solution, exhausting the search space only if the given formula is not satisfiable. This early-termination behavior implies that either nagging or partitioning could just get lucky and quickly encounter a solution, yielding large observed speedup values (this is consistent with the maximum observed speedup values reported in the table). In contrast, the search on unsatisfiable formulae unfolds in a manner more consistent with that of the other search algorithms, since one must exhaust the entire search space before labeling a formula as unsatisfiable. On these problems, nagging's performance clearly dominates that of partitioning by all reported measures.

Of course, summary statistics such as the speedup values just shown obscure the relation between speedup values and problem difficulty: observing a 200-fold performance improvement on a problem that takes several hours to solve on one processor should be more meaningful than observing a similar speedup on a problem that might be solved in just a few milliseconds. To provide a gestalt view of speedup with respect to problem difficulty, we turn to a graphical representation of the data (see Figs. 9–11): these plots show parallel solution time against serial solution time.[14] Interpretation of this kind of plot is relatively straightforward, as datapoints falling below the upper diagonal line are faster in parallel (i.e., have speedup values larger than 1.0), while datapoints falling above the upper diagonal line are faster on one processor. The lower line represents a speedup value equal to the number of processors in use; hence a datapoint falling below this line correspond to superlinear speedups.

Fig. 9 shows results for the *A*/TSP* system. Since identical resource limits are imposed on all serial and parallel trials, doubly-censored datapoints should fall on the diagonal line. Singly-censored datapoints are best understood as datapoints that have been artificially shifted to the left of their true position, because their plotted serial solution times (ordinate) represent lower bounds on their true serial solution times. As is clear from the plot, nagging generally provides some speedup and occasionally provides exceptional speedups, while partitioning is mostly constrained to the region between the two diagonal lines. As expected, the few datapoints that are slower for either nagging or partitioning (i.e., those datapoints corresponding to those speedup values less than 1.0 reported earlier) are relegated to the left hand side of the plot, and represent small problems where the startup costs of parallel execution are not effectively amortized over longer solution times. A similar trend is observed on more difficult (i.e., larger serial solution times) problems, where superlinear speedups are more likely to occur. Aside from the amortization argument, a second factor may also be at work here: one would naturally expect a concomitantly greater payoff by finding a good solution early within a larger search space.

While the mechanism by which a nagging system attains superlinear speedup is clear, it is somewhat less clear how a partitioning system can achieve this kind of performance. To

---

[14] Note that, for clarity, the data are plotted in log–log space, even if this transformation does tend to obscure the relative performance differences on large and small problems: i.e., a 1 unit vertical (alternatively: horizontal) difference on the top half (alternatively: right side) of the plot represents a much larger time interval than an identical 1 unit vertical (alternatively: horizontal) difference on the bottom half (alternatively: left side) of the plot.
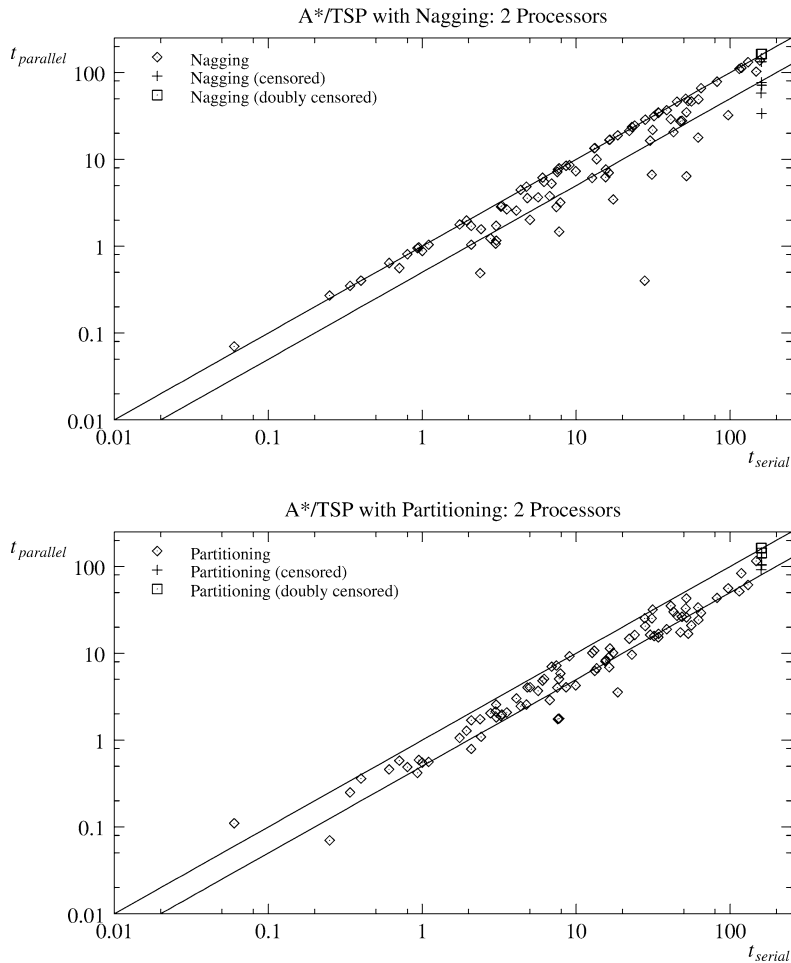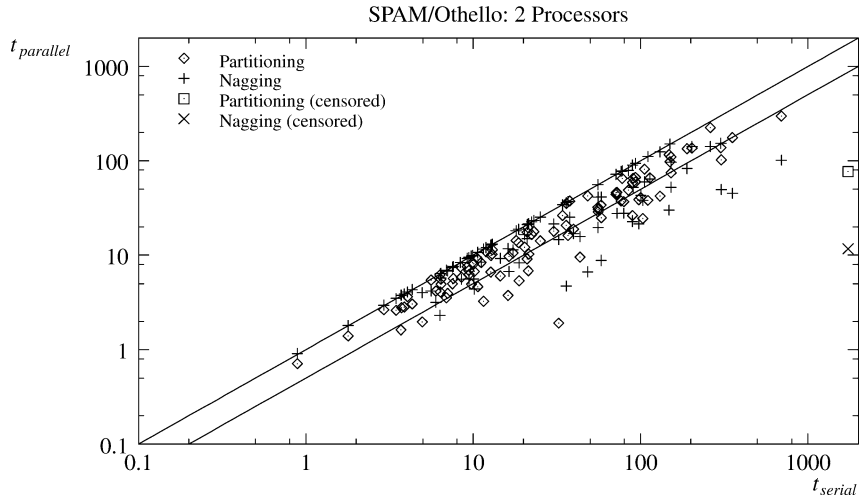
Fig. 9. Two-processor A^*/TSP performance plot in log–log space. Datapoints falling below the upper diagonal line are faster in parallel, while datapoints falling below the lower diagonal line are superlinearly faster in parallel. Doubly-censored datapoints should fall on the diagonal line, while singly-censored datapoints appear artificially displaced to the left of their true positions.

understand how this can happen, recall that the partitioning system implementation tested here differs from the coarse model of the previous section with respect to the asynchronous load balancing policy adopted; as a consequence, multiple partitioning events may occur in the course of solving a single problem. We might therefore attribute those few occasions where partitioning attains superlinear speedups to situations where exceptionally good bounds are found in early partitions, so that subsequent partitioning events can enjoy the benefits in terms of additional pruning. A similar explanation accounts for the occasional superlinear speedups reported by the SPAM/Othello partitioning system (see Fig. 10). Note that, as for A^*/TSP, the SPAM/Othello partitioning system still seems less likely to attain superlinear speedups, especially on larger problems, than does the nagging system, while

SPAM/Othello: 2 Processors

Fig. 10. Two-processor SPAM/Othello performance plot in log–log space. Datapoints falling below the upper diagonal line are faster in parallel, while datapoints falling below the lower diagonal line are superlinearly faster in parallel.

the nagging system on occasion delivers large speedups. Particularly noticeable is the lone censored problem, where because the serial solution time plotted is a lower bound to the true serial solution time, the speedup attained by nagging is *at least* 148.32, as compared with a lower bound speedup of only 22.54 for partitioning. The results shown in Fig. 11 for the Davis–Putnam/3SAT solver show a similar pattern. Recall that both nagging and partitioning can be expected to result in large speedups on satisfiable formulae, due to the algorithm's early-termination behavior. Thus many of the unsatisfiable problems' datapoints lie well in the superlinear speedup zone of the plot. Yet, once again, only nagging seems likely to result in superlinear speedups for (unsatisfiable) problems of any meaningful size.

6.2. Experiment 2

Our analysis of the previous section relied on using strictly solution-equivalent transformation functions. In this experiment, we explore the performance of window narrowing, a non solution-equivalent problem transformation function, in SPAM. Recall that the main idea is that a nagger can artificially restrict its $\alpha\beta$ window in order to gain execution speed at the expense of information: indeed, window narrowing will prune more aggressively, but may not be as informed as solution-equivalent transformations. Our protocol (somewhat arbitrarily) randomly narrows each nagpoint window, while forcing processors corresponding to leaf nodes in the nagging hierarchy to use a unit window, thus essentially performing zero-window search at leaf processors.

The experimental procedure is identical to that of Experiment 1, and the same 100 random Othello problems are used. Note that since we are only using two processors, the nagging processor is always a leaf processor, and is therefore always operating with unit
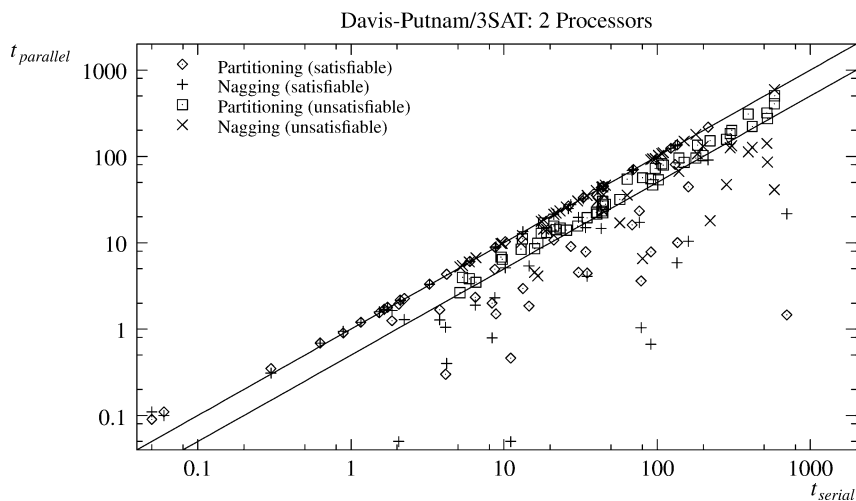
Davis-Putnam/3SAT: 2 Processors

Fig. 11. Two-processor Davis–Putnam/3SAT performance plot in log–log space. Datapoints falling below the upper diagonal line are faster in parallel, while datapoints falling below the lower diagonal line are superlinearly faster in parallel.
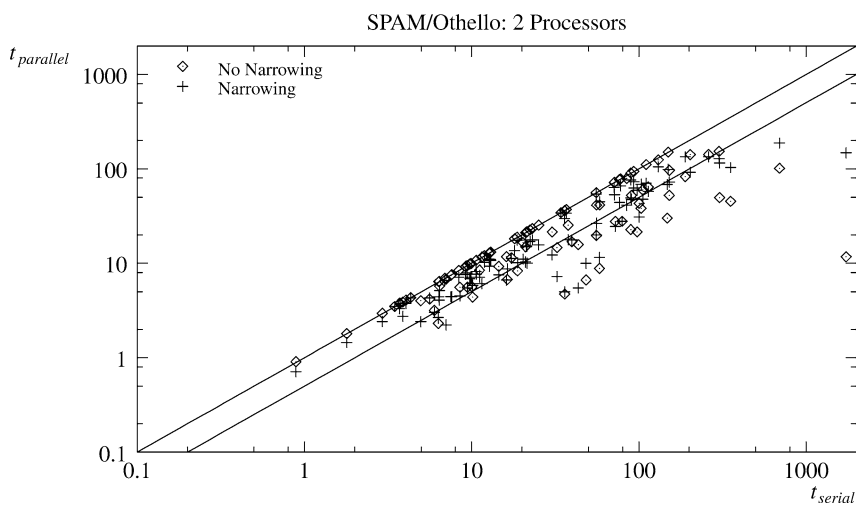
SPAM/Othello: 2 Processors

Fig. 12. Two-processor SPAM/Othello performance plot in log–log space, both with and without the use of the window narrowing transformation. Datapoints falling below the upper diagonal line are faster in parallel, while datapoints falling below the lower diagonal line are superlinearly faster in parallel.

window size. The results, plotted against serial solution times, are shown in Fig. 12 (note that the non-narrowing system data is identical to that shown in Fig. 10).

In addition to graphical comparisons, we can also test, statistically, the null hypothesis "permutation with window narrowing is no faster than permutation alone". If we can reject this null hypothesis, then we can conclude that window narrowing is beneficial. To test

the hypothesis without making any distributional assumptions, we'll use a nonparametric statistic, the paired Wilcoxon signed-ranks test [42]; what such nonparametric statistics may sacrifice in terms of power is more than counterbalanced by their broad applicability. Using 100 paired samples as the input, the paired sign test easily rejects the null hypothesis using the traditional critical value for statistical significance ($p < 0.05$). Of course, statistics notwithstanding, Fig. 12 clearly shows that both transformation functions are doing the job; yet the results shown here do underline the critical importance of the nature of the transformation function used.

6.3. Experiment 3

While both nagging and partitioning are often capable of delivering effective—and, in the case of nagging, often superlinear—speedups, exceptional speedups will not necessarily be the rule for every problem. In this section, we shall examine a problem domain where both nagging and partitioning deliver some speedup, but neither approach manages to produce exceptional results.

Consider the following *team assignment problem*, or TAP, drawn from the sports economics literature:

> Given a collection of N players and T teams, where the ith player has an associated quality value Q_i, find an assignment of players to teams so that there are an equal number of players on each team and the differences between relative team qualities, computed as a function of their constituent player qualities, are minimized.

There are many variants of this problem, depending on the team quality metric used; some more complex variants may involve higher order effects (e.g., individual player quality may be a function of teammate qualities) [41]. For this experiment, however, we'll choose a simple linear metric, so that we are in effect minimizing the sum, over the set of all teams, of the absolute value of the difference between team quality (the sum of player qualities) and the hypothetical average team quality (computed as the product of team size and average player quality). Our goal is to find the best-matched team assignments in terms of team quality; for this particular metric, a perfect solution produces T teams of exactly average quality if such a solution exists. Our solution applies the same A^* search algorithm described in Section 4.2 to the team assignment problem. Formulating an admissible heuristic function that properly bounds the solution value for any partial assignment is not overly difficult; the function used here estimates the potential deviation from the target team quality (i.e., the average player quality times the team size) [30].

The experiment follows the same protocol as Experiment 1 using 100 randomly generated matching problems; the results are shown in Fig. 13.[15] The most striking feature of the plot in Fig. 13 is the extent to which the partitioning system tracks the linear speedup line: only on smaller problems, where solution times are of the same order as the system

---

[15] Random problems were once again generated so that serial solution times ranged between 0.01 seconds, the resolution of the Linux system clock, up to about 5 minutes. The resulting problem set assigned between 15 and 32 players to 2, 3, or 4 teams.
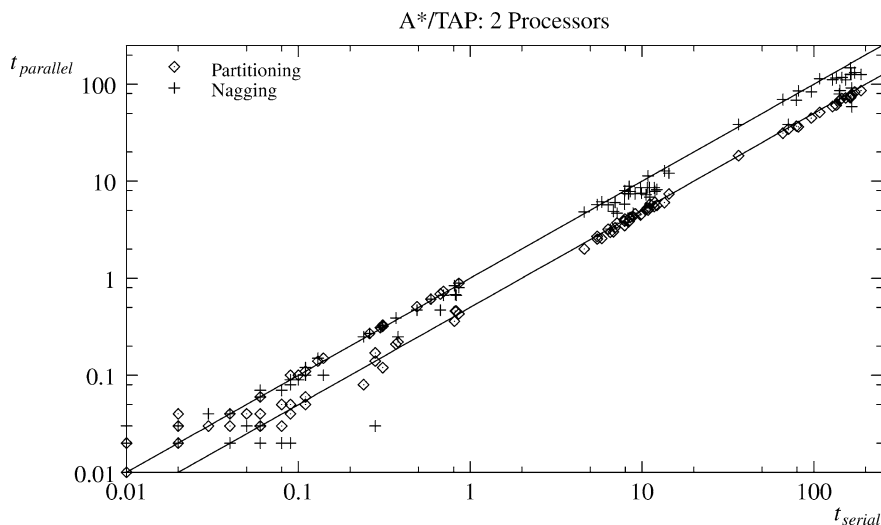
Fig. 13. Two-processor A^*/TAP performance plot. Datapoints falling below the upper diagonal line are faster in parallel, while datapoints falling below the lower diagonal line are superlinearly faster in parallel.

clock resolution, is there any significant deviation from this line. On the other hand, the performance of nagging is decidedly worse than that of partitioning, rarely even attaining linear speedup.

It is tempting to attribute the poor performance of nagging to the use of an inadequate problem transformation function, as permutation of the search space here clearly does not provide additional pruning as for, say, TSP. Yet it is much more likely that the problem lies with the heuristic estimate: if it is not informative (i.e., close to the true additional cost of the partial solution), then A^* search cannot be expected to explore fewer nodes than simple branch-and-bound. One cannot compensate for a poor heuristic estimate with a better problem transformation function. Alternatively, the difficulty may lie with the problem itself. If the cost landscape tends to be populated with many local minima whose values are near that of the global minima, even a highly informative heuristic will not lead to much pruning. As mentioned in Section 4.1, we must acknowledge that some NP-hard problems are harder than others. Unlike 3SAT problems, however, it is possible that TAP problems are simply all uniformly difficult.

6.4. Experiment 4

In this experiment, we provide empirical support for the scalability of nagging, showing how additional processors have a beneficial effect on the performance of the system, and provide direct comparison with the behavior of partitioning. The experimental procedure is like that of Experiment 1, except that we now use 8, 16, 32, or 64 essentially identical

Table 3

CPUs	Nagging speedup					Partitioning speedup				
	N	min	μ	μ^*	max	N	min	μ	μ^*	max
2	92	0.86	2.43	1.51	69.73	91	0.55	1.80	1.68	5.23
64	100	0.96	12.29	4.95	256.22	95	0.38	4.49	3.90	10.21

processors arranged in a hierarchy with branching factor less than or equal to three.[16] In this section, we will focus on results obtained with 64 processors.[17]

We turn first to some simple descriptive statistics. Recall that in Experiment 1 using only two processors, there were 8 problems left unsolved by nagging, and 9 problems left unsolved by partitioning (these correspond to the doubly-censored datapoints of Fig. 9). When 64 processors are applied, the nagging system solves all 100 problem optimally, while the partitioning system still leaves five unsolved problems. So, at least qualitatively, the performance of nagging exceeds that of partitioning. We can also use observed speedup values to help quantify this trend (see Table 3; as before, doubly-censored datapoints are excluded). Direct comparison of both μ and μ^* confirms the advantage of nagging in this experiment; indeed, the maximum observed speedup for nagging exceeds the maximum observed speedup for partitioning by more than a factor of twenty. Moreover, the values given in the table are fully consistent with the argument first advanced in Section 5.3, that is, that the probability of obtaining a superlinear speedup is higher for nagging than for partitioning (here, the nagging system produces superlinear speedups on at least three of 100 problems—and possibly more, given the amount of censoring observed—while partitioning fails to produce any superlinear speedup at all).

One troubling fact is that the observed mean speedups μ and μ^* are significantly less than N, the number of processors employed. In our analytical model, the predicted μ^* values—while still less than N—were significantly more in line with N. We can attribute this discrepancy to two differences. First, the model of Section 5 is a coarse model, where all processors engage in a single nagging episode on the root node of the search, while the experimental model allows repeated nagging episodes applied at internal nodes of the search process. Second, and more to the point, the analytical model had all $N-1$ naggers reporting directly to the single master search process, while the experiment allowed no more than three processors to nag the master directly (the remaining $N-4$ processors were used to recursively nag the naggers). That the NICE hierarchy limits each daemon to only three descendents is quite arbitrary; while increasing the branching factor of the hierarchy raises the communication overhead incurred by the master, one must balance the increased overhead on the performance benefits obtained. Note that the tradeoff is complicated, since the optimal configuration may differ depending on the search algorithm or even the problem instance. In any case, adaptive configuration of the NICE hierarchy is still an area we are actively exploring.

---

[16] The actual hierarchy depends on the NICE resource management daemon, and is constantly changing in response to local system load and availability.

[17] For the record, in our tests, both nagging and partitioning scale smoothly from 2 to 64 processors.
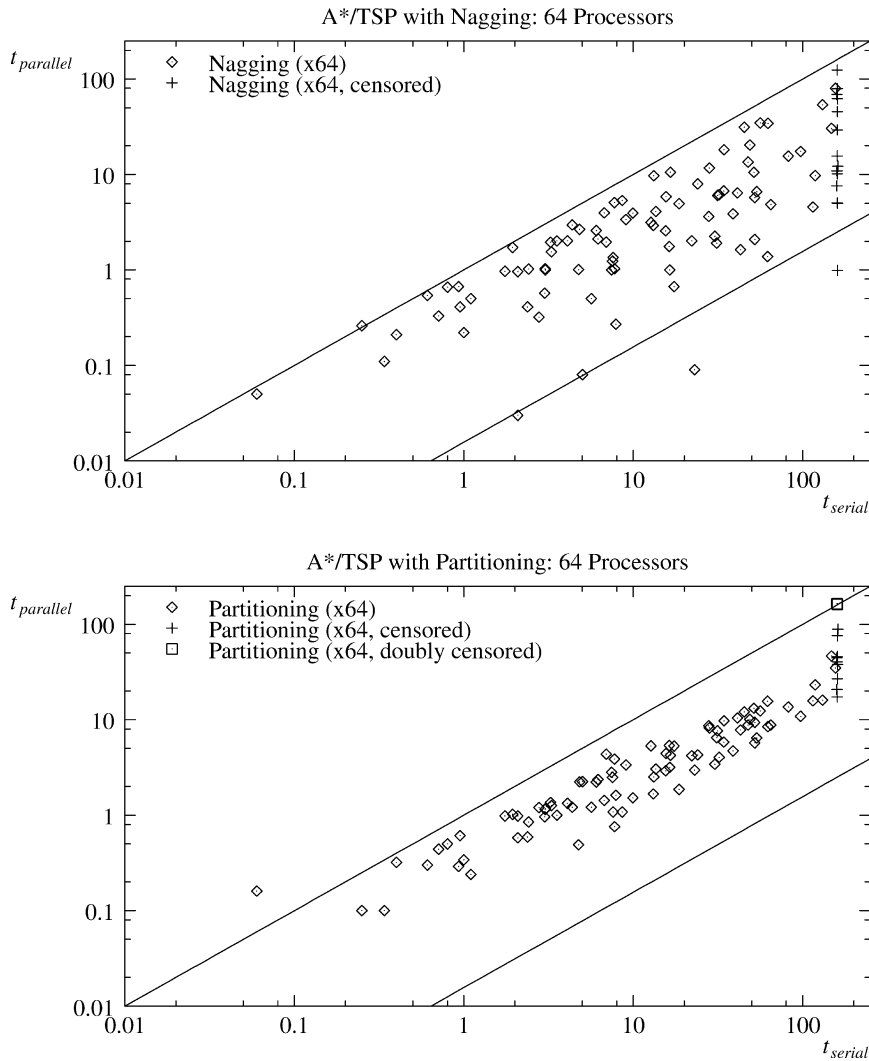
Fig. 14. Sixty-four processor A^*/TSP performance plots for nagging (top) and partitioning (bottom); compare with two processor A^*/TSP performance plots given in Fig. 9. Datapoints falling below the upper diagonal line are faster in parallel, while datapoints falling below the lower diagonal line are superlinearly faster in parallel.

Fig. 14 presents a graphical view of the same data, and can be compared directly with the plots of Fig. 9 to confirm the descriptive statistics outlined above: nagging indeed appears more effective than partitioning in applying additional processors to reduce solution time. Not surprisingly, this observation appears more striking on larger problems, where the cost of initializing additional processors is readily amortized over longer solution times. We conjecture that this trend extends to still larger problems; indeed, the fact that nagging

solves all problems optimally while partitioning still leaves five unsolved problems within the prespecified resource bound is consistent with this conjecture.

7. Conclusion

Nagging is a paradigm for parallel search in a heterogeneous distributed computing environment. It is applicable to a broad range of different artificial intelligence search algorithms, and scales easily to a large number of processors. We have shown, using an analytical performance model, how nagging is often a superior approach to the more traditional partitioning strategy commonly employed to parallelize search. We have also presented empirical results that confirm the predictions of our analytical model and support our claims regarding both the performance and scalability of nagging across several different domains and search algorithms.

We are currently working on a number of refinements to nagging. First, inspired by the empirical and analytical results reported here, we have already experimented with randomly mixing nagging and partitioning within the same search. The idea is that since nagging and partitioning appear to be complementary, one should actively manage a mixture of approaches in order to more effectively guide the use of computational resources. We are currently focusing on how to decide whether a new event should be a nagging event or a partitioning event. Based on the analytic model of Section 5, it should be possible to use statistical evidence obtained in the course of a problem solving episode to decide how best to use an idle processor on a particular problem. This decision might well change over the course of the search: for example, early events might be primarily nagging events and later events might be primarily partitioning events. The basic idea is to let information about this particular problem solving process guide the best application of processor power over the course of the search process.

We are also working on extensions to the NICE infrastructure itself. We are experimenting with better hierarchy-formation and restructuring algorithms in order to better apply the available computational resources to a given problem. We are also looking at cryptographic certification techniques for distributing new NICE-enabled applications to processors within the NICE hierarchy.

Much of this work is being performed in the context of an extraordinarily challenging computational biology application. Over the last two years, we have been working on HOPS, an *ab initio* distributed hybrid optimization protein structure prediction engine (see, e.g., [10]). HOPS is large, interdisciplinary, project involving faculty and students from biochemistry, computer science, operations research and applied mathematics. It combines a distributed search (using both nagging and partitioning) over a discrete space of protein conformations with traditional continuous optimization techniques (e.g., nonlinearly constraint nonlinear programming, interior point methods, etc.) to find the energetically most favorable conformation of a specified protein according to an energy model of our own design. Given the sheer size of these search problems, HOPS is a perfect example of the kind of application where nagging's distinguishing features—effectiveness, scalability, and fault tolerance—should shine.

Acknowledgements

References

[1] D. Applegate, R. Bixby, W. Cook, and V. Chvátal, On the solution of traveling salesman problems, Technical Report 98744, Center for Research on Parallel Computation, Rice University, Houston, TX, July 1998.

[2] A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM Users Guide and Reference Manual, Oak Ridge National Laboratory, Oak Ridge, TN, 1994.

[3] M. Brockington, J. Schaeffer, APHID: Asynchronous parallel game-tree search, J. Parallel Distributed Comput. 60 (2) (2000) 247–273.

[4] D. Cook, R.C. Varnell, Adaptive parallel iterative deepening search, J. Artificial Intelligence Res. 9 (1998) 139–166.

[5] S. Cook, The complexity of theorem-proving procedures, in: Proc. 3rd Annual ACM Symposium on the Theory of Computing, 1971, pp. 151–158.

[6] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Comm. ACM 5 (7) (1962) 394–397.

[7] W. Ertel, Performance analysis of competitive or-parallel theorem proving, Technische Universität München, FKI-162-91, 1992.

[8] R. Feldmann, P. Mysliwietz, B. Monien, Game tree search on a massively parallel system, in: H. van den Herik, I. Herschberg, J. Uiterwijk (Eds.), Advances in Computer Chess VII, University of Limburg, Maastricht, the Netherlands, 1994, pp. 203–218.

[9] C. Ferguson, R.E. Korf, Distributed tree search and its application to alpha-beta pruning, in: Proc. AAAI-88, St. Paul, MN, 1988, pp. 128–132.

[10] S.L. Forman, Torsion angle selection and emergent non-local secondary structure in protein structure prediction, Ph.D. Thesis, Department of Mathematics, The University of Iowa, Iowa City, IA, August 2001.

[11] I. Foster, C. Kesselman, Globus: A metacomputing infrastructure toolkit, Internat. J. Supercomput. Appl. High Performance Comput. 11 (2) (1997) 115–128.

[12] C. Gomes, B. Selman, N. Crato, H. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, J. Automat. Reasoning 24 (1–2) (2000) 67–100.

[13] A. Grama, V. Kumar, Parallel search algorithms for discrete optimization problems, ORSA J. Comput. 7 (4) (1995) 365–385.

[14] A. Grama, V. Kumar, State of the art in parallel search techniques for discrete optimization problems, IEEE Trans. Knowledge Data Engrg. 11 (1) (1999) 28–35.

[15] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message-passing interface standard, Parallel Comput. 22 (6) (1996) 789–828.

[16] J. Gu, P.W. Purdom, J. Franco, B.W. Wah, Algorithms for the satisfiability (SAT) problem: A survey, in: Satisfiability Problem: Theory and Applications, in: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence, RI, 1997, pp. 19–151.

[17] C. Joerg, B. Kuszmaul, Massively parallel chess, in: Proc. Third DIMACS Parallel Implementation Challenge, Rutgers University, Rutgers, NJ, 1994.

[18] D.E. Knuth, R.W. Moore, An analysis of alpha-beta pruning, Artificial Intelligence 6 (4) (1975) 293–326.

[19] T.H. Lai, S. Sahni, Anomalies in parallel branch-and-bound algorithms, Comm. ACM 27 (4) (1984) 594–602.

[20] E. Lawler, D. Wood, Branch and bound methods: A survey, Oper. Res. 14 (4) (1966) 699–719.

[21] E. Lee, Statistical Methods for Survival Data Analysis, Second Edition, Wiley, New York, 1992.

[22] M. Litzkow, M. Livny, M. Mutka, Condor: A hunter of idle workstations, in: Proc. Eighth Conference on Distributed Computing Systems, San Jose, CA, 1988.

[23] A. Mahanti, C.J. Daniels, A SIMD approach to parallel heuristic search, Artificial Intelligence 60 (2) (1993) 243–282.

[24] N. Mann, R. Schafer, N. Singpurwalla, Methods for Statistical Analysis of Reliability and Life Data, Wiley, New York, 1974.

[25] W. Meeker, L. Escobar, Statistical Methods for Reliability Data, Wiley, New York, 1998.

[26] D. Mitchell, B. Selman, H. Levesque, Hard and easy distributions of SAT problems, in: Proc. AAAI-92, San Jose, CA, 1992, pp. 459–465.

[27] A. Newell, J. Shaw, H. Simon, Chess playing programs and the problem of complexity, IBM J. Res. Development 2 (1958) 320–335.

[28] N. Nilsson, Problem-Solving Methods in Artificial Intelligence, McGraw Hill, New York, 1971.

[29] P. Norvig, Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp, Morgan Kaufmann, San Mateo, CA, 1992.

[30] H.J. Paarsch, A.M. Segre, Extending the computational horizon: Effective distributed resource-bounded computation for intractable problems, in: Proc. Fifth International Conference of the Society for Computational Economics, 1999.

[31] C. Powley, C. Ferguson, R.E. Korf, Depth-first heuristic search on a SIMD machine, Artificial Intelligence 60 (2) (1993) 199–242.

[32] A. Reinefeld, V. Schnecke, Work-load balancing in highly parallel depth-first search, in: Proc. 1994 Scalable High-Performance Computing Conference, 1994, pp. 773–780.

[33] G. Reinelt, The Traveling Salesman: Computational Solutions for TSP Applications, Springer, Berlin, 1994.

[34] A.M. Segre, C.P. Elkan, A. Russell, A critical look at experimental evaluations of EBL, Machine Learning 6 (2) (1991) 183–196.

[35] A.M. Segre, D.B. Sturgill, Using hundreds of workstations to solve first-order logic problems, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 187–192.

[36] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proc. AAAI-92, San Jose, CA, 1992, pp. 440–446.

[37] B. Selman, D. Mitchell, H. Levesque, Generating hard satisfiability problems, Artificial Intelligence 81 (1996) 17–29.

[38] M.A. Stephens, EDF statistics for goodness of fit and some comparisons, J. Amer. Statist. Soc. 69 (1974) 720–727.

[39] D.B. Sturgill, A.M. Segre, Nagging: A distributed adversarial search-pruning technique applied to first-order logic, J. Automat. Reasoning 19 (3) (1997) 347–376.

[40] D.B. Sturgill, A.M. Segre, A novel asynchronous parallelization scheme for first-order logic, in: Proc. Twelfth Conference on Automated Deduction, 1994, pp. 484–498.

[41] C. Whittinghill, Social choice and resource allocation in a professional sports league, Ph.D. Thesis, Department of Economics, The University of Iowa, Iowa City, IA, 1999.

[42] F. Wilcoxon, Individual comparisons by ranking methods, Biometrics 1 (1945) 80–83.

[43] W. Zhang, State-Space Search: Algorithms, Complexity, Extensions, and Applications, Springer, Berlin, 1999.